Real-Time Operating Systems (RTOS)

C.A.Maynard (c)2021
clive.maynard@monash.edu

Reference: Jean J. Labrosse "The 10-minute Guide to RTOS"

Electronic Design Magazine, June 1994

Introduction

Real-time systems are characterised by the fact that severe consequences can result if logical as well as timing correctness properties of the system are not met.

A real-time multitasking application is a system in which several time critical activities must be processed simultaneously.

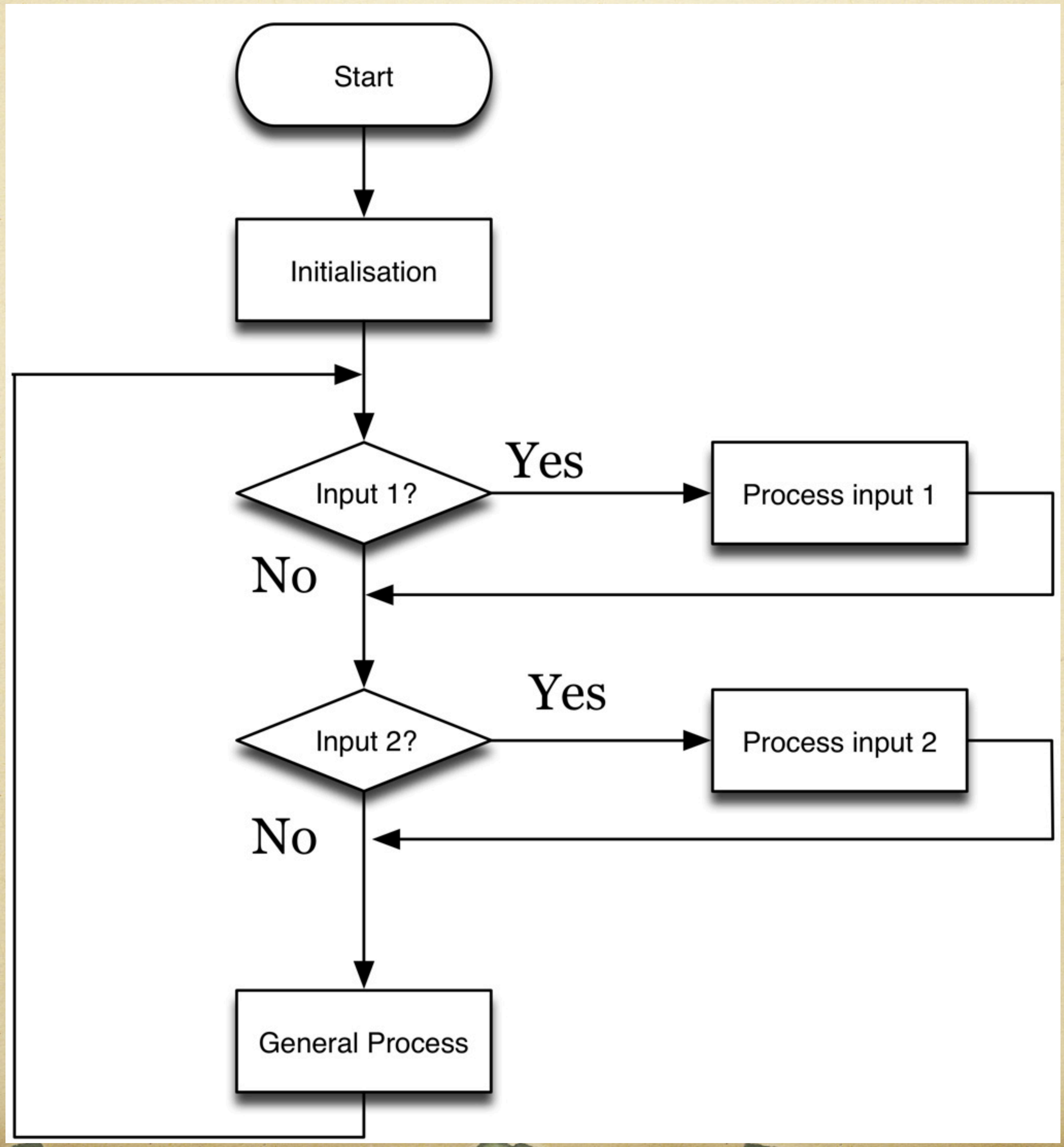
A real-time multitasking kernel (also called a real-time operating system, RTOS) is software which ensures that time critical events are processed as efficiently as possible.

The use of a RTOS generally simplifies the design process of a system by allowing the application to be divided into multiple independent elements called tasks (or processes).

A Polling Loop without an RTOS required.

- Polling is one of the techniques for getting data into a system.
- The simplest process is a loop of execution where each possible source of data is checked and if there is new data the necessary code is executed to deal with the information.
- In an embedded system the loop continues to execute while the program is running. An infinite loop.

A PROGRAM FLOWCHART



THE POLLING PROCESS

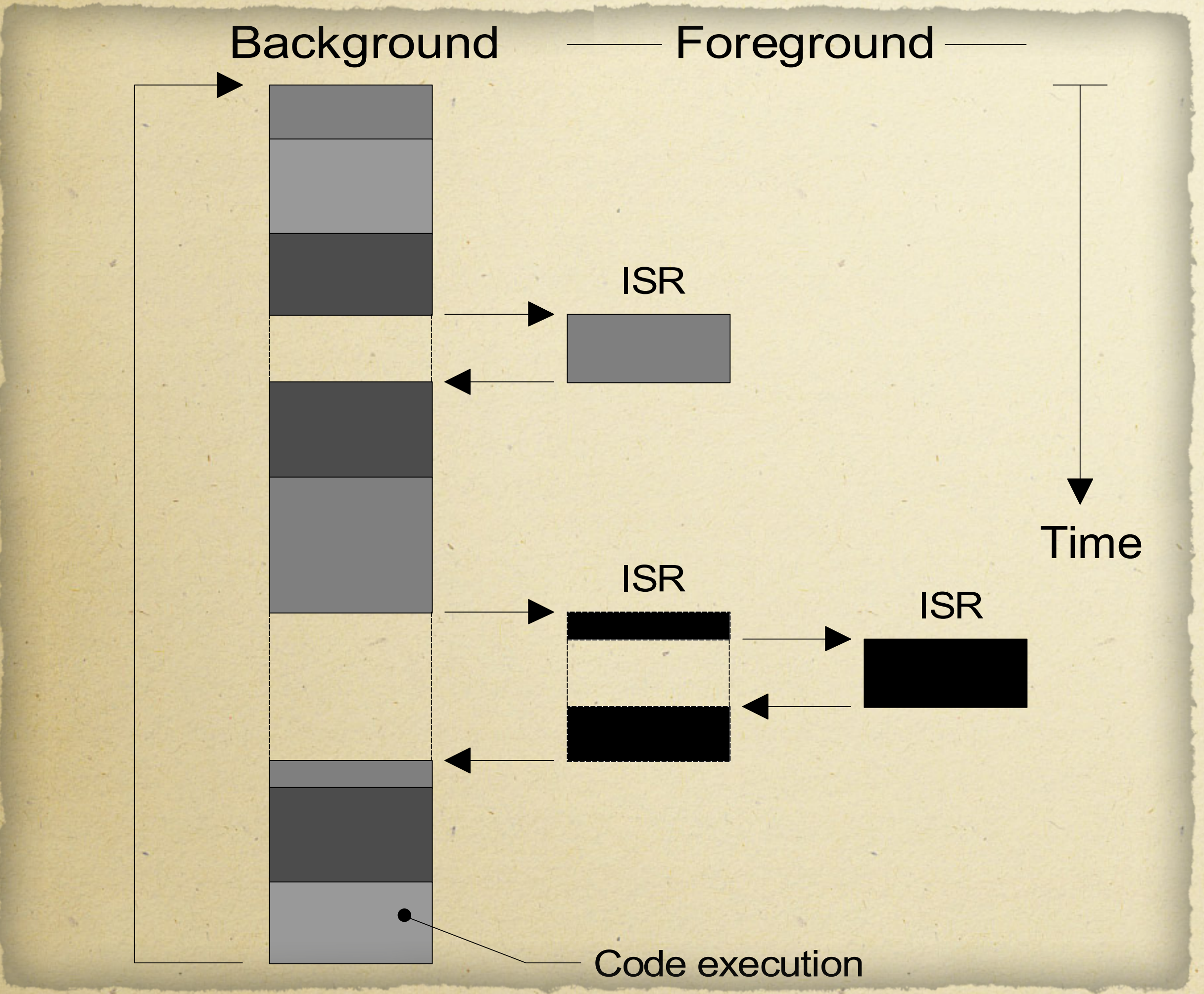
- If a new value is available from the input source then execute the required process.
- Go to the next input and repeat as necessary.
- After the last input perform any general processing required then go back and start again.
- The restart of the polling process MAY be subject to a time constraint which has not been shown in the flowchart.

Can we improve with Interrupts?

- Foreground/Background processing.

- An application consists of an infinite loop which calls application modules to perform the desired operations.
- The modules are executed sequentially (background) with interrupt service routines (ISRs) handling asynchronous events (foreground).

Foreground/Background



Critical operations must be performed by the ISRs to ensure that they are dealt with in a timely fashion. Because of this, ISRs have a tendency to take longer than they should.

Information for a background module made available by an ISR is not processed until the background routine gets its turn to execute.

The latency in this case depends on how long the background loop takes to execute.

Real-Time Kernels

Multitasking .
Multitasking is the process of scheduling and switching the CPU between several tasks; a single CPU switches its attention between several sequential tasks.

Multitasking provides the ability to structure your application into a set of small, dedicated tasks that share the processor. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity which is inherent in real-time applications.

Real-time kernels can make application programs easier to design and maintain.

A task is a simple program which thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem.

Kernel

The kernel is the part of the multitasking system responsible for the management of tasks and communication between tasks.

When the kernel decides to run a different task, it simply saves the current task's context (CPU registers) onto the current task's stack; each task is assigned its own stack area in memory.

Once this operation is performed, the new task's context is restored from its stack area and then, execution of the new task's code is resumed. This process is called a context switch or a task switch.

The current top of stack for each task, along with other information, is stored in a data structure called the Task Control Block (TCB).

A TCB is assigned to each task when the task is created and is managed by the RTOS

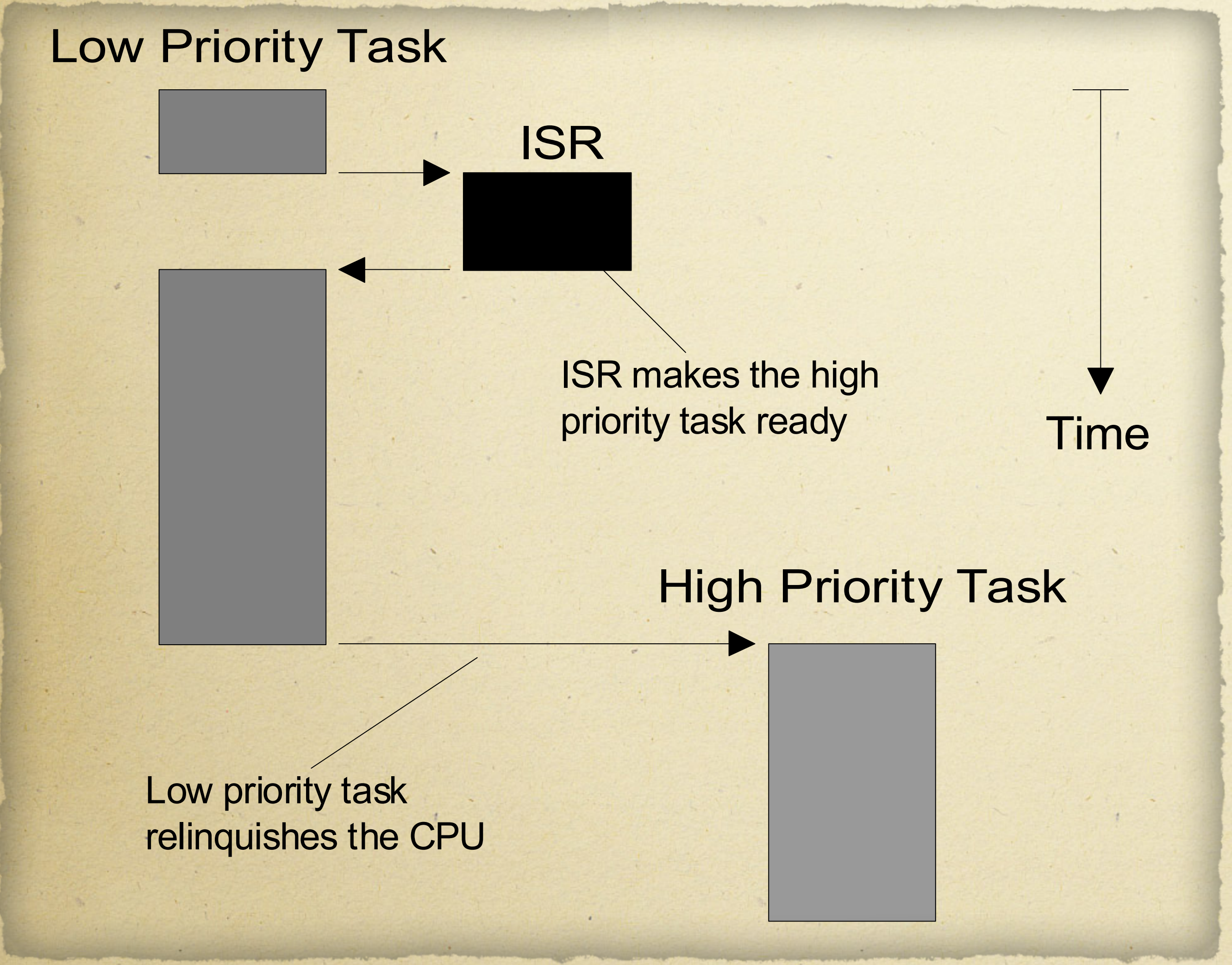
Interrupts & The Scheduler

An important issue in real-time systems is the time required to respond to an interrupt and to actually start executing user code to handle the interrupt. All RTOSs disable interrupts when manipulating critical sections of code. The longer a RTOS disables interrupts, the higher the interrupt latency. RTOSs generally disable interrupts for less than about 50 uS but obviously, the lower the better. The code that services an interrupt is called an **Interrupt Service Routine (ISR)**.

The scheduler, also called the dispatcher, is the part of the kernel which is responsible for determining which task will run next and when. Most real-time kernels are priority based; each task is assigned a priority based on its importance. Establishing the priority for each task is application specific. In a priority based kernel, control of the CPU will always be given to the highest priority task ready to run. When the highest priority task gets the CPU, however, depends on the type of scheduler used.

There are two types of schedulers: **non-preemptive and preemptive**.

Non-preemptive Scheduling


Low Priority Task
ISR
ISR makes the high priority task ready
Time
High Priority Task
Low priority task relinquishes the CPU

Non-preemptive schedulers require that each task does something to explicitly give up control of the CPU.

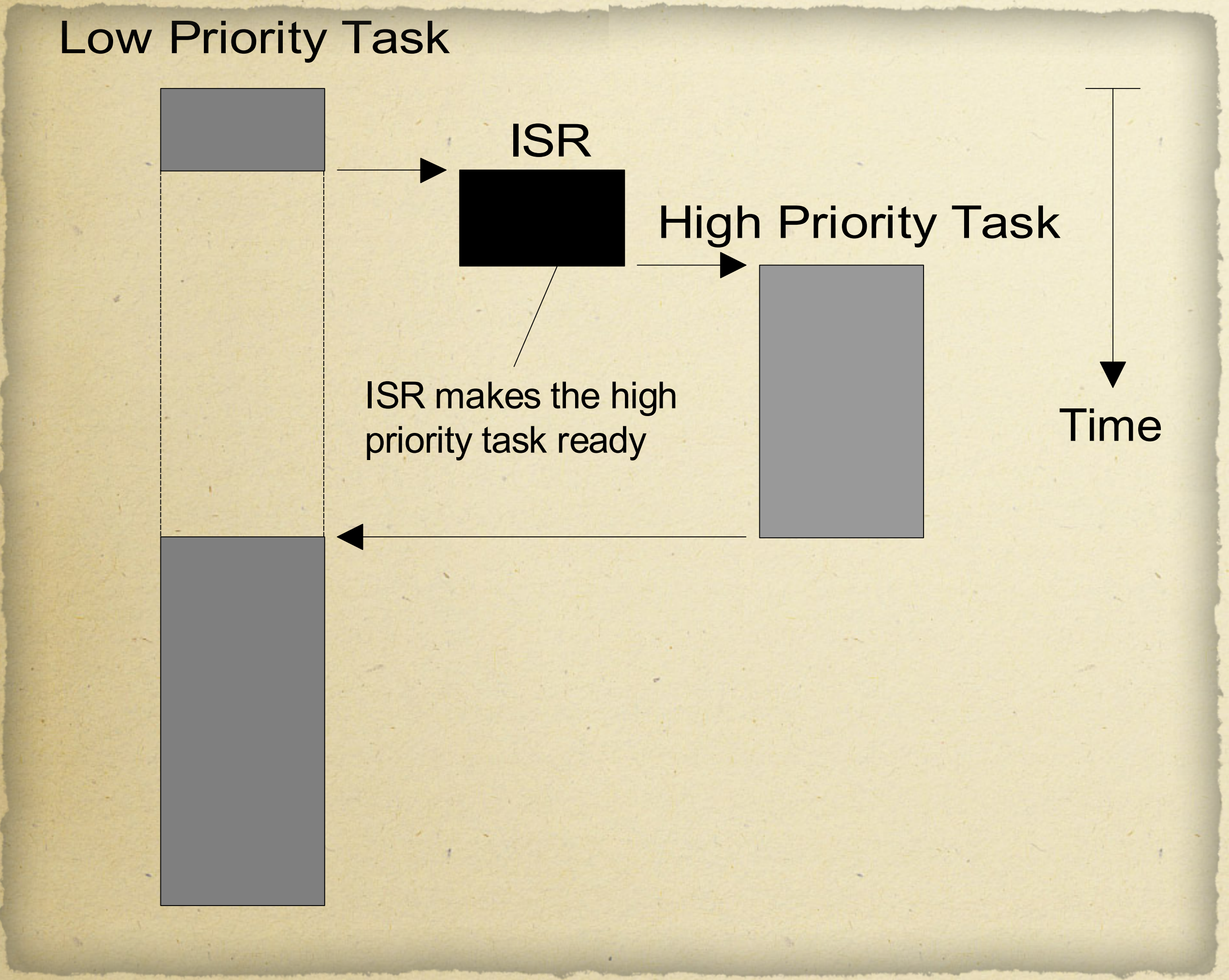
To maintain the illusion of concurrency, this process must be done frequently. Non-preemptive scheduling is also called **cooperative multitasking**; tasks cooperate with each other to gain control of the CPU.

When a task relinquishes the CPU, the kernel executes the code of the next most important task that is ready to run. Asynchronous events are still handled by ISRs. An ISR can make a higher priority task ready to run but the ISR always returns to the interrupted task.

The new higher priority task will gain control of the CPU only when the current task voluntarily gives up the CPU.

Latency of a non-preemptive scheduler is much lower than with Foreground/ Background; latency is now given by the time of the longest task.

Preemptive Scheduling



In a preemptive kernel, when an event makes a higher priority task ready to run, the current task is immediately suspended and the higher priority task is given control of the CPU.

If an ISR makes a higher priority task ready to run, the interrupted task is suspended and the new higher priority task is resumed.

Most real-time systems employ preemptive schedulers because they are more responsive.

Reentrancy

A **reentrant** function is a function which can be used by more than one task without fear of data corruption.

Conversely, a **non-reentrant** function is a function which cannot be shared by more than one task unless mutual exclusion to the function is ensured by either using a semaphore (described later) or, by disabling interrupts during critical sections of code.

A reentrant function can be interrupted at any time and resumed at a later time without loss of data.

Reentrant functions either use local variables (CPU registers or variables on the stack) or protect their data when global variables are used.

Compilers specifically designed for embedded software will generally provide reentrant run-time libraries. Non- preemptive schedulers do not require reentrant functions unless a function is shared between a task and an ISR. Preemptive schedulers require you to have reentrant functions if the functions are shared by more than one task.

Kernel Services

Real-time kernels provide services to your application. One of the most common services provided by a kernel is the management of semaphores. A semaphore is a protocol mechanism used to control access to a shared resource (mutual exclusion), signal the occurrence of an event or allow two tasks to synchronise their activities. A semaphore is basically a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner. A suspended task consumes little or no CPU time.

Kernels also provide time related services allowing any task to delay itself for an integral number of system clock ticks. A clock tick typically occurs every 10 to 200 mS depending on the application requirements.

It is sometimes desirable for a task or an ISR to communicate information (i.e. send messages) to another task. This is called intertask communications and services for sending and receiving messages are generally provided by most kernels. The two most common kernel services for sending messages are the message mailbox and message queue.

A message mailbox, also called a message exchange is typically a pointer size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Both the sending and receiving task will agree as to what the pointer is actually pointing to; they will agree on the message content.

A message queue is used to send more than one message to a task. A message queue is basically an array of mailboxes.

Processes (or Tasks)

- A process is a unique execution of independent activity except for well defined communications with other processes.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Why multiple processes?

- Processes help us manage timing complexity:
- multiple rates
  - multimedia
  - automotive
- asynchronous input
  - user interfaces
  - communication systems

Example: Home alone

- Last weekend, my wife had dinner with her friends…
- My tasks for the evening:
  - Cook dinner
  - Eat dinner
  - Watch TV

Example: Home alone

- The wildcard!!!

Example: Home alone

- Last weekend, my wife had dinner with her friends …
  - My tasks for the evening:
- Cook dinner
- Eat dinner
- Watch TV
- Play
- Clean up toys
- Feed Luca
- Clean Luca

Life without processes/OS

- Peel/chop onions and garlic
  - If (dirty diaper)
  - Clean diaper
  - Fry onions, garlic & mince.

If (dirty diaper)
- Clean diaper

If (screaming Luca)
- Play with Luca
- If (burning food)
  - Turn-off stove

Life with processes/OS

Process cook
- Peel/chop onions and garlic
- Fry onions, garlic & mince.
- Add wine, carrots,…
- Create process (stir)

Process stir
- Stir food

Process diaper
- Clean diaper

Process play

Process eat:
- lift up food
- Move food to mouth
- Chew

Process feed Luca
- lift up food
- Move food to mouth
- Wait for mouth to open

- …

Life with processes/OS

```
Main () {
        create_process (adult());
        create_process (Luca());
}

adult() {
        create_process(eat);
        create_process(dinner);
        …
}
```

```
Luca() {
        create_process (diaper);
        create_process (feed);
        …
}
```

Adapted from chapter 6 by Wolf "Computers as Components: principles of embedded system design" Morgan Kaufmann © .

Running Realtime Examples

- For the next few lectures we will be using runnable examples of multitasking operations.
- The source code is available in Moodle and so are the precompiled examples on Windows.
- It is my intention to take advantage of the current recorded lecture scenario to suggest which one to run, suggest you suspend the lecture, expect you to run the example then come back to the lecture for the follow up discussion. This should work reasonably well.
- BUT if you use a Mac what do you do?

Run Windows or Windows programs on your Mac

On a Mac, you have several options for installing software that allows you to run Windows and Windows applications:

- To dual-boot between macOS and Windows, use Apple's Boot Camp . This approach provides the most compatibility with Windows software and peripherals, but does not allow you to run Windows and macOS applications at the same time.
- To run Windows in a virtual machine within macOS, use Parallels Desktop , VMware Fusion , or VirtualBox . This method will allow you to run Mac and Windows applications concurrently.
- To run Windows programs without having to install Windows itself, use a Windows compatibility layer, such as CrossOver Mac . This option typically offers good functionality for a limited set of Windows applications.